Git in Four Weeks - Week 1
Revision 1.0
05/25/21

Lab 1 - Creating and Exploring a Git Repository and Managing Content

In this lab, we'll create an empty Git repository on your local disk and stage and commit content into it.

Prerequisites

To complete this and all future labs in the course, you must have a working version of Git installed. We assume 2.0 or higher for the version. (To see which version you have, you can run **git --version**) If you don't have a working version of Git installed, then you should install it now.

Steps

1.) On your local disk, **create a new directory** and change (**cd**) into it. (This will be the directory we work in unless otherwise specified).

```
$ mkdir some-dir

$ cd some-dir
```

2.) In the new directory, initialize a new repo by running the command:

```
$ git init
```

This command created a new git repository skeleton in a subdirectory named "**.git**" under the current directory - as indicated by the output message from the command. This means that you're now able to start using other Git commands in the current directory.

3.) Tell Git who you are by setting your basic identification configuration settings with the following commands (Note the double dashes before "global" since we are spelling out the option. Also, values only require quotes if there is a space in the value.)

```
$ git config --global user.name "first-name last-name"

$ git config --global user.email your-email-address
```

4.) Now let's create some content to put through the Git workflow. Note that for purposes of these initial labs, we just need files to work with - we don't really care what's in them. We can "cheat" and just echo something into a file via the ">" operator. In fact, the output of any command could be used to put content into a file via the ">" operator. Of course, if you prefer, you can certainly create files via your favorite editor instead.

Create two files - contents and names don't matter.

$ echo content > ***file1-name***

$ echo content > ***file2-name***

5.) Stage the files with the add command. (If you prefer you can add each separate file explicitly rather than use the ".")

$ git add .

Note: If you see any messages about end of line conversions, you can ignore them.

6.) Now commit the files. You can use whatever commit message (comment) you want. Note the single hyphen/dash before the short form of the option.

$ git commit -m "*commit-message***"**

7.) Notice the output you get. There is the branch name - the default branch - **master**, followed by an indicator that this was the first (root) commit and then the first few characters of the SHA1 for the commit.

8.) Edit one of the files. (We can just use the ">>" to append something to the file's content.)

$ echo more >> ***file1-name***

9.) Stage and commit the file with the shortcut. Note the combined short options "-am" for "-a" + "
-m".

$ git commit -am "*commit-message***"**

END OF LAB

Lab 2 - Tracking Content through the File Status Lifecycle

In this lab, we'll work through some simple examples of updating files in a Local Environment and viewing the status and differences between the various levels along the way.

Prerequisites

This lab assumes that you have done Lab 1: Creating and Exploring a Git Repository and Managing Content. You should start out in the same directory as that lab.

Steps

1.) Starting in the same directory as you used for Lab 1, run the status command or the short form to see how it looks when you have no changes to be staged or committed.

$ git status (or git status –s)

2.) Create a new file and view the status.

$ echo content > *file3-name*

$ git status (or git status -s)

Is the file tracked or untracked?

Answer: It's untracked - we haven't added the initial version to Git yet.

3.) Stage the file and check status

$ git add . (or git add *file3-name*)

$ git status (git status –s if you want)

Is the file tracked or untracked? What does "Changes to be committed" mean?

(Answers: The file is now tracked - we've added the initial version to Git. "Changes to be committed" implies files exist in the Staging Area and the next step for them is to be committed into the Local Repository.)

4.) Edit the same file again in your Working Directory and check the status.

$ echo change > *file3-name*

$ git status

Why do we see two?

Where is the version that's listed as "Changes to be committed"? (Working Directory, Staging Area, or Local Repository)

Where is the version that's listed as "Changes not staged for commit"? (Working Directory, Staging Area, or Local Repository)

(Answers: We see two because there is one version of the same file in the Working Directory and another version in the Staging Area.

The version that's listed as "Changes to be committed" is in the Staging Area. The phrase implies that this version's "next step" or "next level for promotion" is to the Local Repository via a commit.

The version that's listed as "Changes not staged for commit" is in the Working Directory. The phrase implies that this version's "next step" or "next level for promotion" is to the Staging Area, since it's currently "not staged".)

5.) Do a diff between the version in the Working Directory and the version in the Staging Area.

$ git diff

6.) Go ahead and commit and do another status check.

$ git commit -m "*commit-message*"

$ git status

Which version did we commit – the one in the Staging Area or the one in the Working Directory? (Hint: Which one is left – shows up in the status? Note the "Changes not staged for commit" part of the status message.)

(Answer: The version in the Staging Area was the one committed. The content goes through the Staging Area and then into the Local Repository.)

7.) Stage the modified file you have in your Working Directory and do a status check.

$ git add .

$ git status

8.) Edit the file in the Working Directory one more time and do a status check.

$ echo "change 2" > ***file3-name***

$ git status

At this point, we have a version of the same file in the Local Repository (the one we committed in step 6), a version in the Staging Area (the one we staged in step 7), and a version in the Working Directory (step 8).

9.) Diff the version in the working area against the version in the Staging Area.

$ git diff

10.) Diff the version in the Staging Area against the version in the Local Repository.

$ git diff --staged (or git diff --cached) (note the -- is a double -)

11.) Diff the version in the working area against the version in the Local Repository (the one we committed earlier).

$ git diff HEAD

12.) Commit using the shortcut.

$ git commit -am "*commit-message*"

Which version got committed – the one in the Working Directory or the one in the Staging Area?

(Answer: Since we used the -am shortcut, the version from the Working Directory was staged (over the previous version in the Staging Area) and then that version was committed into the Local Repository.)

13.) Check the status one more time.

$ git status

Notice the output - we're back to a clean Working Directory - Git has the latest versions of everything we've updated.

END OF LAB

Lab 3 - Working with Changes Over Time and Using Tags

In this lab, we'll work through some simple examples of using the Git log commands to see the flexibility it offers as well as creating an alias to help simplify using it. We'll also look at how to tag commits to have another way to reference them.

Prerequisites

This lab assumes that you have done Lab 2: Tracking Content through the File Status Lifecycle. You should start out in the same directory as that lab.

Steps

1.) Starting in the same directory as you used for Lab 3, let's first make another change to the repository to make the history more interesting. Add a line to the first file you committed into the repository and then stage and commit. Note that you can use the shortcut here.

```
$ echo new >> file1-name
$ git commit -am "commit-message"
```

2.) Now, take a look at the history we have so far in our small repository. To do this we just run the log command. (In some terminals your history may be longer than the screen and need to hit a key to continue. If you are paging through log output and want to end the listing, hit the "q" key.)

```
$ git log
```

3.) Often when looking at Git history information, users will only want to see the first line of each entry - the "subject line". This is why it is important to make that first line meaningful in a real-life use of Git.

To see only the first line of each log message, you can use the --oneline option. Try it now.

```
$ git log --oneline
```

4.) Let's try a more complex version of the log command that includes selected pieces of history information formatted in a specific way. Be careful of your typing - note the colon after "format", the double hyphens, and the double quotes.

```
$ git log --pretty=format:"%h %ad|%s %d[%an]" --date=short
```

Press **Enter** to see this execute.

5.) Since this is a bit much to type, let's create an alias to simplify running this command. We do this by configuring the alias name to stand for the command and its options. Enter the following, paying attention to the punctuation (double hypens, colon, vertical bars, single and double quotes, etc.) Note this command spans multiple lines - copy and paste may not work as expected.

```
$ git config --global alias.hist "log --
pretty=format:'%h %ad | %s%d [%an] ' --date=short"
```

6.) Now run your new hist alias. You should see the same output as the original log command from step 3. If you encounter any problems, go back and double-check what you typed in step 4.

```
$ git hist
```

7.) We can also use the log command (and our hist alias) on individual files. Pick one of your files and run the hist alias against it.

```
$ git hist some-existing-file
```

8.) We're interested in seeing the differences between a couple of the revisions. But there are no version numbers. How do we pick revisions?

(Answer: We pick revisions via the SHA1 (hash) values (first 7 bytes are enough). It's the first column in the hist output.)

9.) Run the git hist alias again and find the SHA1 values of the earliest and latest lines in the history. (Yours will of course be different from mine in the example below.)

```
$ git hist
```

```
   latest (top) -> 1db49cf 2016-08-20 | add a line (HEAD -> master) [Brent Laster]
                   ece66a5 2016-08-20 | committing another change [Brent Laster]
                   8103190 2016-08-20 | update [Brent Laster]
                   581c751 2016-08-20 | another update [Brent Laster]
earliest (bottom)-> c6a82d2 2016-08-20 | first commit [Brent Laster]
```

10.) We can use these SHA1 values similarly to how we might use version numbers in other systems. Let's see the history between our earliest and latest commits. To do this, we'll run the hist alias and specify the range of values using the SHA1 values. Execute the command below, substituting the appropriate SHA1 values from the history in your repository.

```
$ git hist earliest-SHA1..latest-SHA1
```

11.) You should see a similar history as you saw previously. One thing to note here is you don't see the original (first) commit. This is because when specifying ranges via the ".." syntax, Git defines that as essentially everything after the first revision. Note that you can also run this against an individual file. Try the command below with your SHA1 values and the first file you added in the repository.

```
$ git diff earliest-SHA1..latest-SHA1 file1-name
```

12.) This is useful, but finding and typing SHA1 values each time for operations like this can be cumbersome. To simplify this, we can use tags to point to commits, and then use those tag names instead of the SHA1 values in commands. Let's create tags for the earliest and latest commits in our repository. We'll use the tags "first" and "last" respectively. The commands are below.

Format: git tag *tagname hash*

```
$ git tag first earliest-SHA1
$ git tag last latest-SHA1
```

13.) Now that we have the tags, we can use them anyplace we used the SHA1 values before. Try out the hist alias with the tags.

```
$ git hist first..last
```

14.) You may not have thought about it, but this is giving us the history for all of the files in the repository. This is because a tag applies to an entire commit - not a specific file in the commit. To see this more clearly, add the --name-only option to the command and run it again.

```
$ git hist first..last --name-only
```

15.) What do we do if we only want to do an operation using a tag for one file? The answer is to simply add that filename onto the command. Try out the example below.

```
$ git hist first..last --name-only file1-name
```

END OF LAB